UNIVERSITY OF ALBERTA

Pointer Analysis

CMPUT 497/500
Foundations of Program Analysis

Karim Ali
@karimhamdanali

Previously

- Call graph = calling relationships at compile time
- Sound (no missing edges)
- Precise (few spurious edges)
- On-the-fly call graph construction

Previously

- Andersen-style (e.g., SPARK)
- Rapid Type Analysis (RTA)
  - single points-to set for the program
- Variable Type Analysis (VTA)
  - field-based, simplify SCC, no OTF
- Steensgaard-style
  - equality-based (not subset-based)

Today

other variants of points-to

Points-to Analysis
vs
Alias Analysis

Points-to Analysis

Points-to Analysis

Points-to Analysis

points-to(v) = {o_1, o_2, ... }

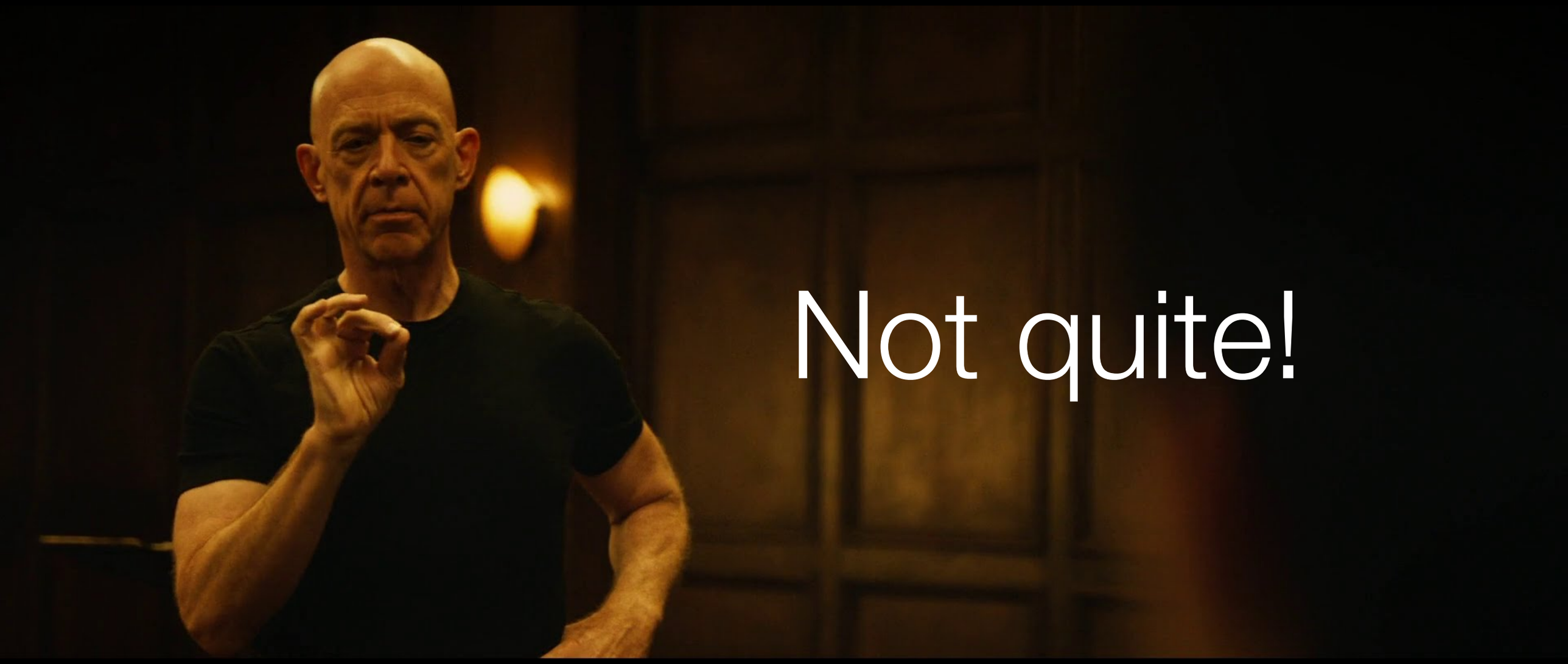
Alias Analysis

Alias Analysis

V_1 V_2

Alias Analysis

$$alias(v_1, v_2) = true/false$$


Not quite!

Points-to vs Alias

allocation sites

$$\text{points-to}(v) = \{a_1, a_2, \ldots\}$$

may/must

Points-to vs Alias

$$\text{points-to}(v) = \{a_1, a_2, \ldots\}$$

$$\text{may-alias}(v_1, v_2) = \text{true/false}$$

$$\text{must-alias}(v_1, v_2) = \text{true/false}$$

May-Alias vs Must-Alias

```
a = new A();
if(..) {
  b = a;
}
c = new C();
d = c;
```

```
may-alias(a,b) =   true
must-alias(a,b) =  false
may-alias(a,c) =   false
must-alias(c,d) =   true
```

May-Alias vs Must-Alias

```
a = new A();
if(..) {
  b = a;
}
c = new C();
d = c;
```

```
may-alias(a,b) =   true
must-alias(a,b) =  false
may-alias(a,c) =   false
must-alias(c,d) =  true
```

Must-alias is typically associated with control flow!

Must-Alias => Flow-Sensitive?

```
    b = null;
    d = null;
s1: a = new A();
    if(..) {
      b = a;
    }
s2: c = new C();
    b = c;
s3: d = a;
```

must-alias(a,s1,d,s2) = false
must-alias(a,s1,d,s3) = true

must-alias(b,s2,c,s2) = false
must-alias(b,s2,c,s3) = false

Must-Alias => Flow-Sensitive?

```
    b = null;      must-alias(a,d) =      false
    d = null;      must-alias(a,d) =      true
s1: a = new A();
    if(..) {       must-alias(b,c) =      false
      b = a;       must-alias(b,c) =      false
    }
s2: c = new C();
    b = c;
s3: d = a;
```

Have to be conservative!

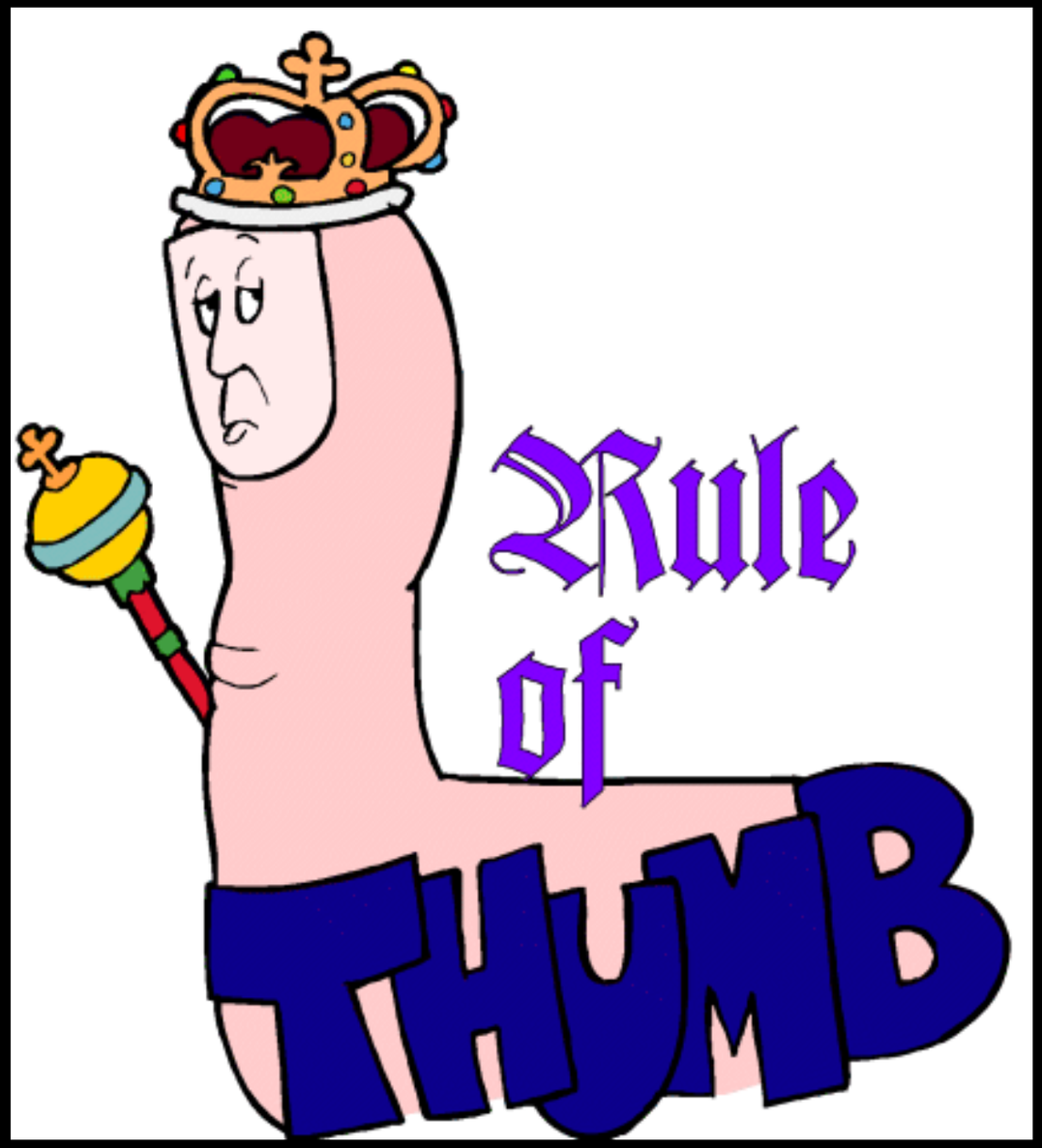
Must-Alias => Flow-Sensitive?

```
    b = null;
    d = null;
s1: a = new A();
    if(..) {
      b = a;
    }
s2: c = new C();
    b = c;
s3: d = a;
```

must-alias(a,d) = false

must-alias(b,c) = false

- Must-X analyses must be flow-sensitive.
- May-X analyses may be flow-(in)sensitive.



Points-to as Alias

Points-to as Alias

$$alias(v_1, v_2) =$$

$$points\text{-}to(v_1) \cap points\text{-}to(v_2) \neq \emptyset$$

When to use Alias Analysis?

Using Alias Analysis

```
void readProp(String id, String d) {
  String s = Properties.read(id);
  if(s==null) s = d;
  return s;
}
```

Assume you can't analyze Properties.read()
(e.g., native method, unknown library)

Using Alias Analysis

```
void readProp(String id, String d) {
  String s = Properties.read(id);
  if(s==null) s = d;
  return s;
}
```

```
may-alias(s, d) = true
```

Using Alias Analysis

```
void readProp(String id, String d) {
  String s = Properties.read(id);
  if(s==null) s = d;
  return s;
}

may-alias(s, d) = true

points-to(s) = ∅
points-to(s) = any object
```

unsound

imprecise

Using Alias Analysis

```
void readProp(String id, String d) {
  String s = Properties.read(id);
  if(s==null) s = d;
  return s;
}
```

```
may-alias(s, d) = true
points-to(s) = ?
points-to(d) = ?
```

```
may-alias(s, default) =
  points-to(s) ∩ points-to(default) ≠ ∅
```

For Incomplete Programs

- Associating variables with allocation sites is either unsound or imprecise (i.e., points-to)

- Alias analysis is better suited, because it can reason about the relationship between variables without caring about which objects they point to

“Direct” Alias Analysis

“Direct” Alias Analysis

```
b = a;
```

```
may-alias(b,a) = true
```

```
c = b;
```

```
may-alias(c,b) = true
```

“Direct” Alias Analysis

```
a = null;
```

may-alias(b,a) = true

```
b = a;
```

may-alias(c,b) = true

```
c = b;
```

usually ignored!

$$\text{may-alias}(v_1,v_2) = \text{may-alias-or-both-null}(v_1,v_2)$$

When to use

Points-to Analysis?

Using Points-to Analysis

```
a1: l = new LinkedList();
    l.clear();
```

```
points-to(l) = { a1 }
type-of(points-to(l)) = { LinkedList }
```

`l.clear()` can only invoke `LinkedList.clear()`

For method de-virtualization,
alias analysis has almost no use

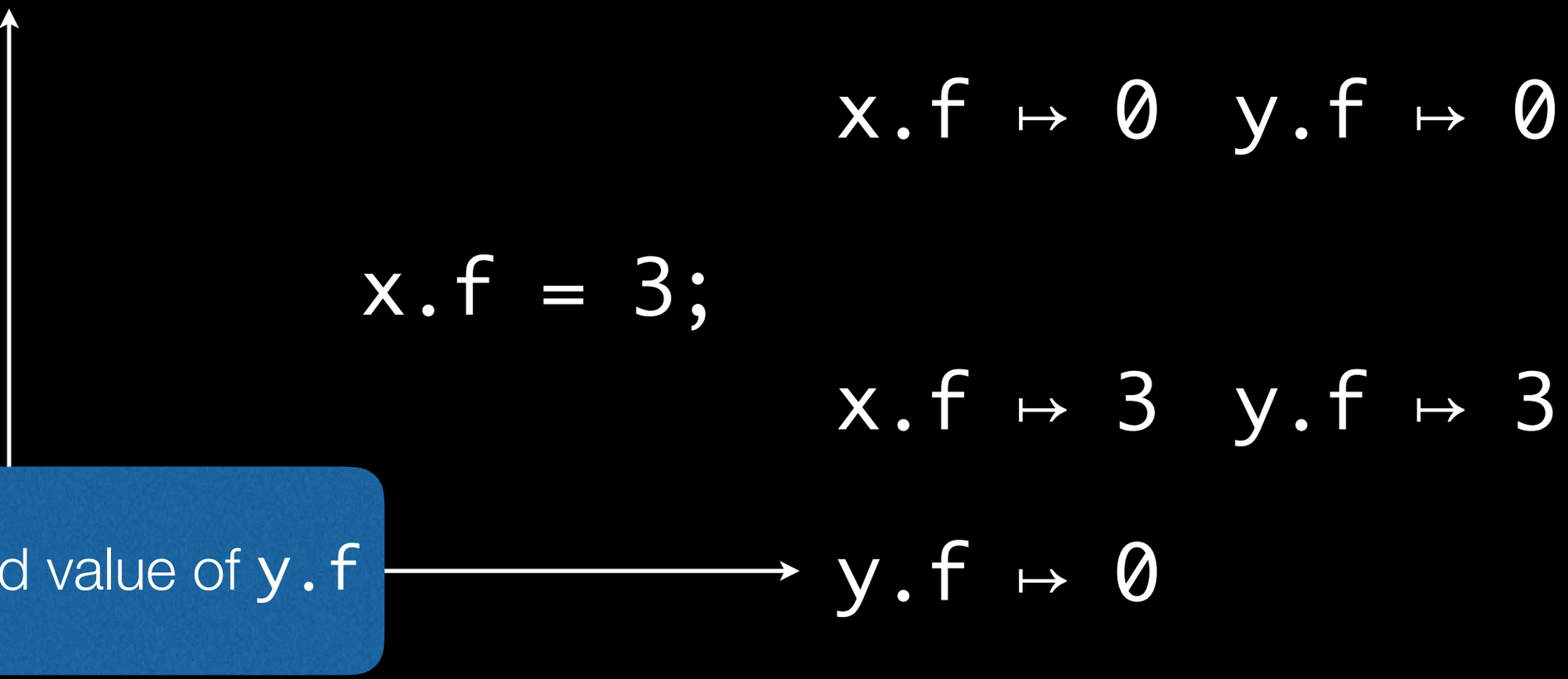
Weak Updates
vs
Strong Updates

Weak Updates

- Doable if only `may-alias` info is available
- Retain previous info, and add to it
- Cannot *kill* old info (leads to unsound results)

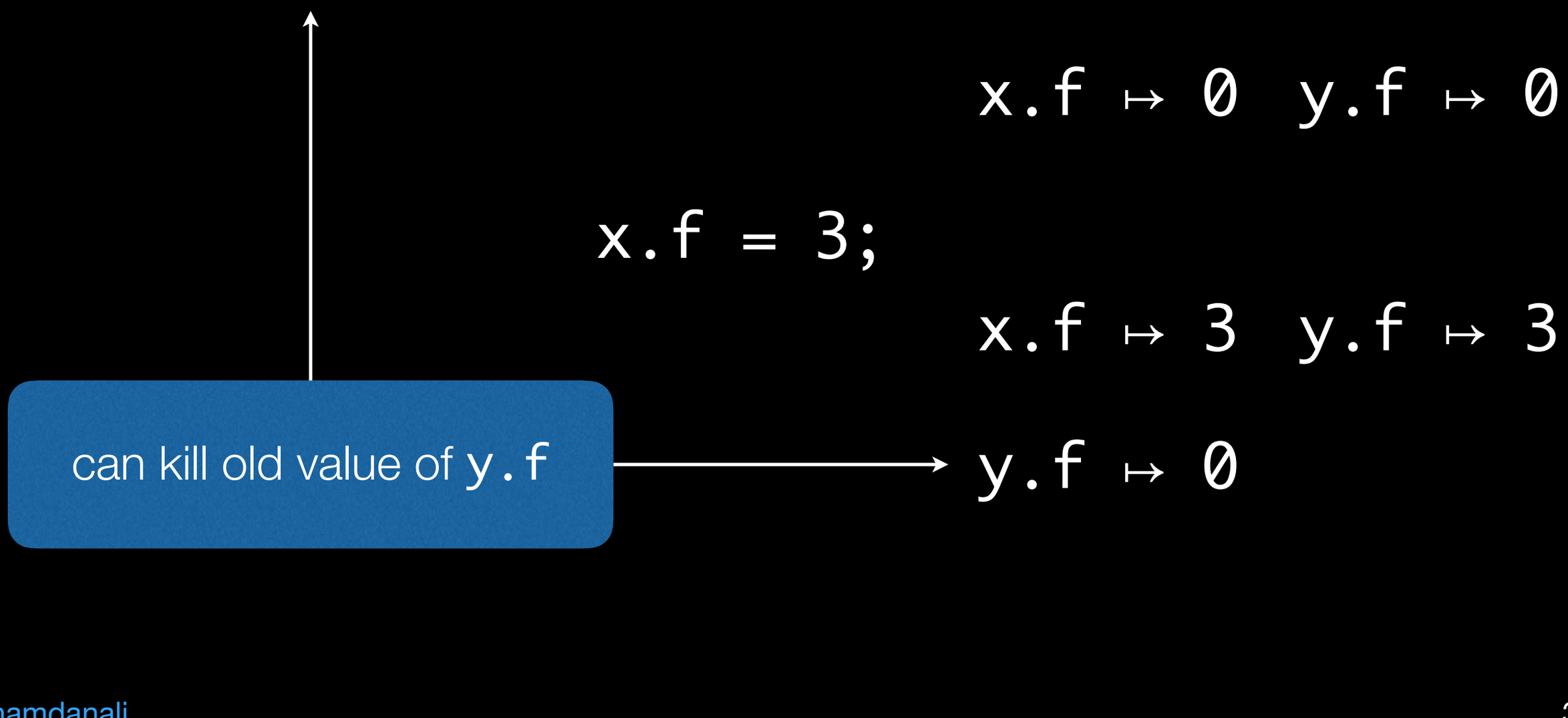
Weak Updates

- constant propagation
- variables initialized to 0
- only `may-alias(x,y)` is known



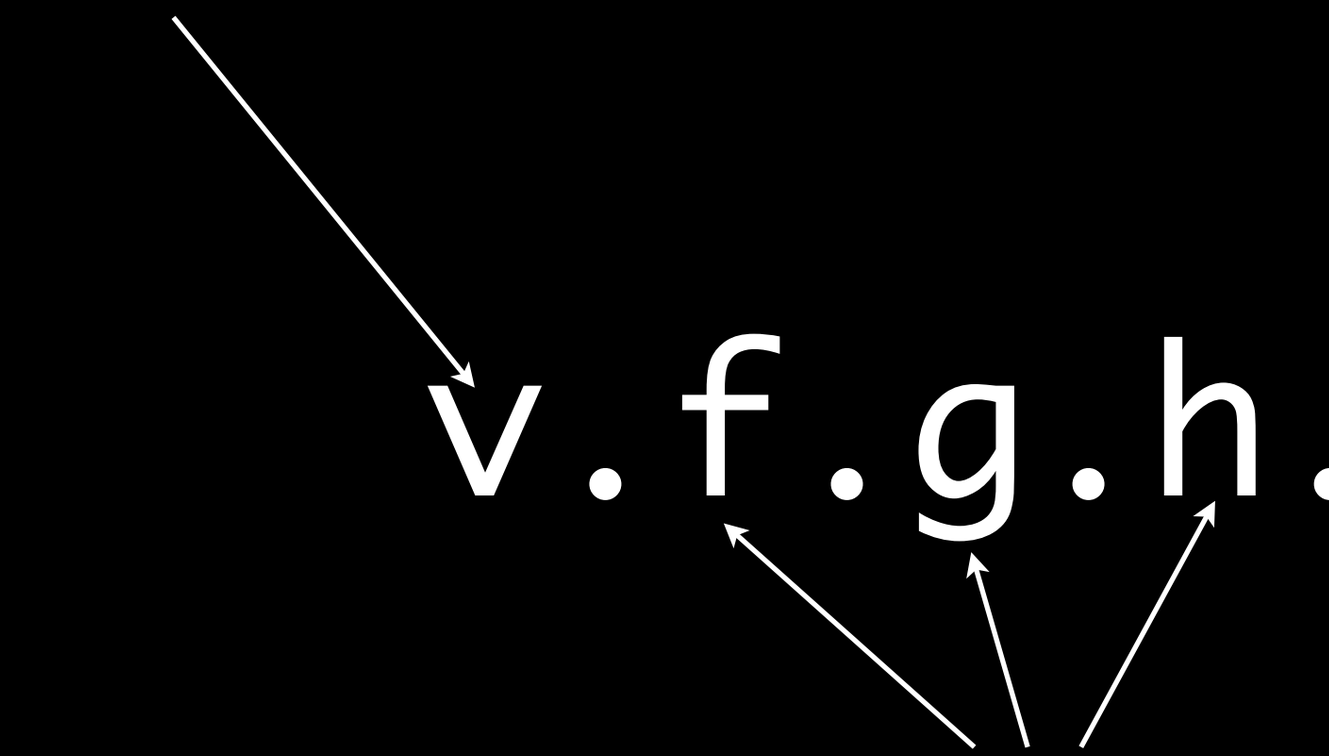
Strong Updates

- constant propagation
- variables initialized to 0
- `must-alias(x,y)` is known



Access Paths

local variable



field accesses

Access Paths as Object Descriptors

```
x = new X();
```
{ x }

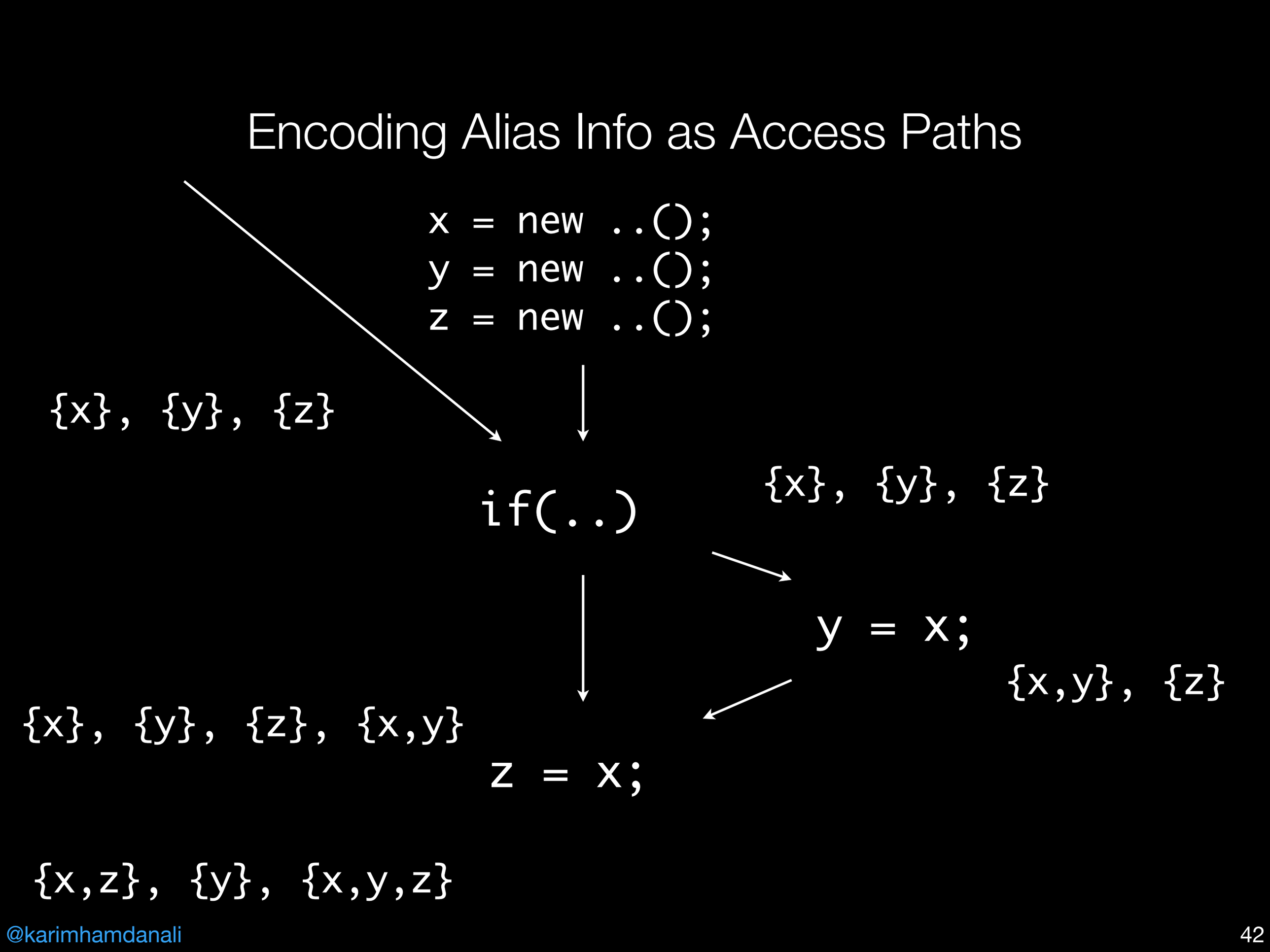
```
a.f = x;
```
{ x, a.f }

```
b.g = a.f;
```
{ x, a.f, b.g }

```
c.h = b;
```
{ x, a.f, b.g, c.h.g }

{x}, {x,y}

Encoding Alias Info as Access Paths



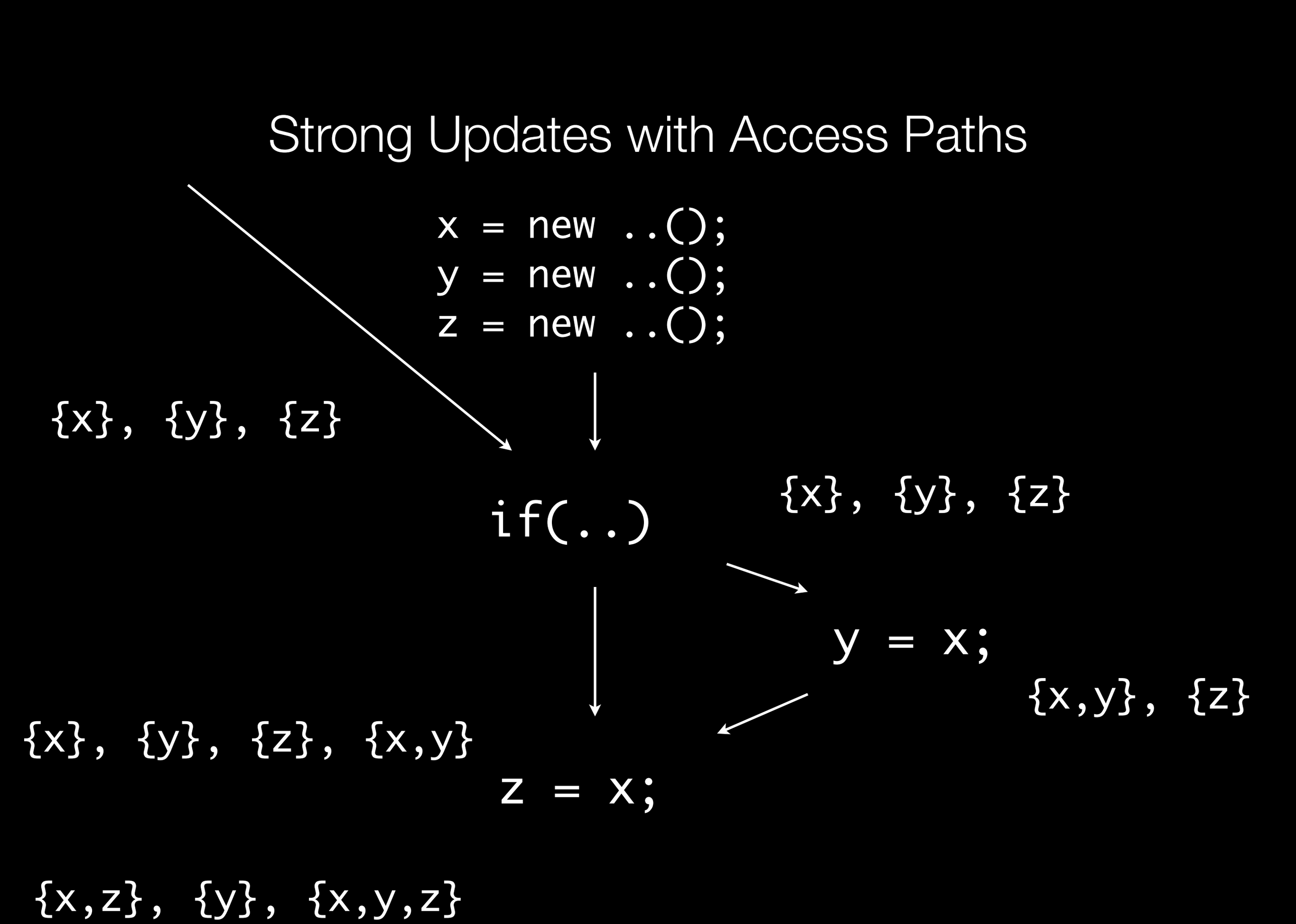
Encoding Alias Info as Access Paths

- `may-alias(x,y)` if there is a set containing both `x` and `y`
- `must-alias(x,y)` if each set that contains `x` also contains `y`

```
 may-alias(x,y) = true
must-alias(x,z) = true          {x,z}, {x,y,z}
must-alias(x,y) = false
```

Strong Updates with Access Paths

Strong Updates with Access Paths



Strong Updates with Access Paths

- constant propagation
- variables initialized to 0

```
{x}, {y}, {z}, {x,y}
```

```
{x}.f ↦ 0, {y}.f ↦ 0,
{z}.f ↦ 0, {x,y}.f ↦ 0
```

```
z = x;
```

```
{x,z}, {y}, {x,y,z}
```

```
{x,z}.f ↦ 0, {y}.f ↦ 0,
{x,y,z}.f ↦ 0
```

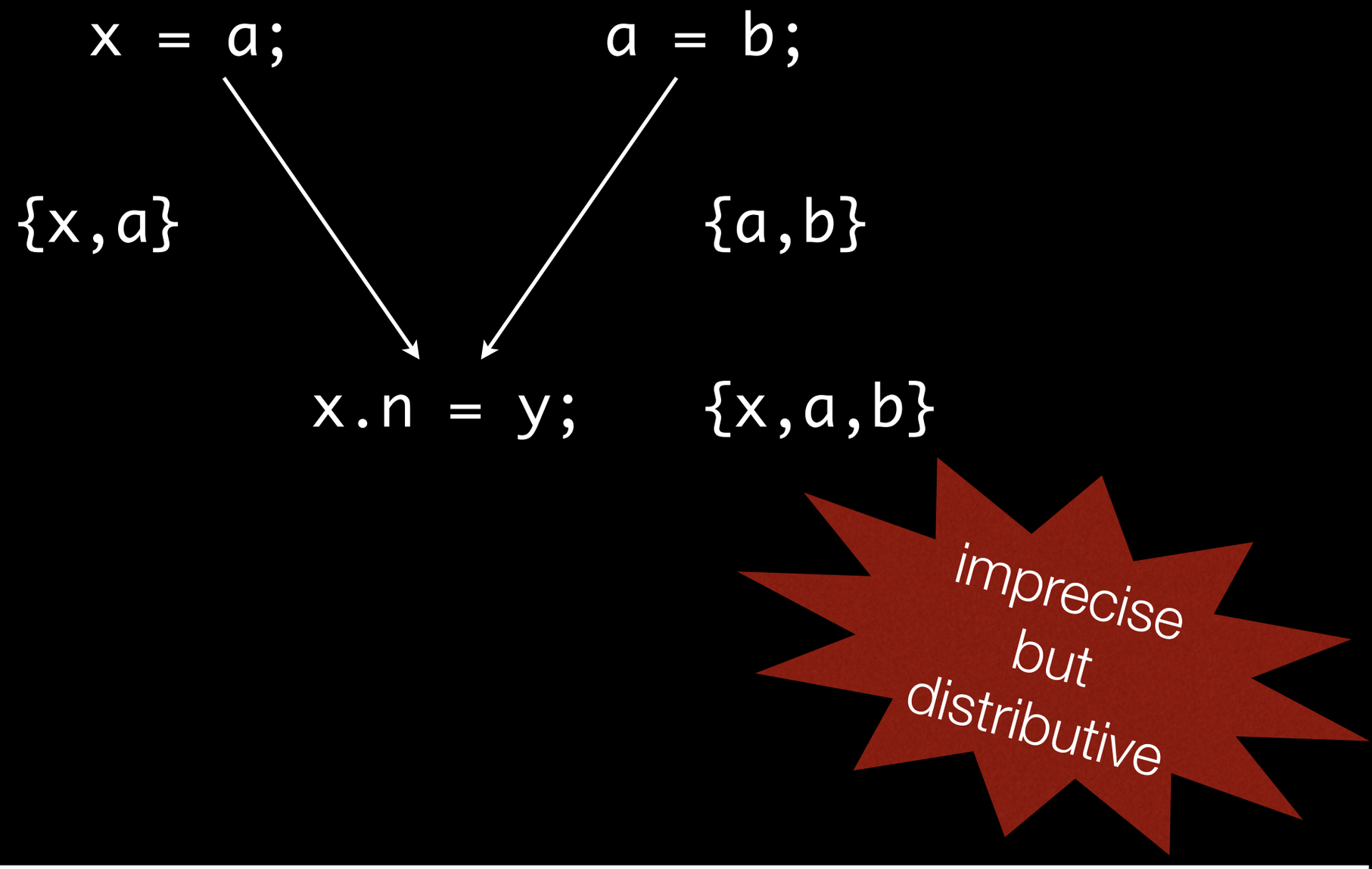
```
x.f = 3;
```

Strong Update

```
{x,z}, {y}, {x,y,z}
```

Pointer Analysis
&
Distributivity

Pointer Analysis & Distributivity



Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural Pointer Alias Analysis. ACM Transactions on Programming Languages and Systems 21, 4 (July 1999), 848-894.

Pointer Analysis & Distributivity

- In general, pointer analysis is **not** distributive
- Merging first yields different results than merging later
- $f(x \sqcup y) \neq f(x) \sqcup f(y)$

Summary

- Certain Points-to analyses can be used to also answer alias-analysis queries
  - Advantage: re-use points-to analysis results
- Must-alias => flow-sensitive setting
- Strong update requires must-alias information
- Flow-sensitive points-to analysis is not distributive

Next

- IFDS framework